

Negace, bilingvální opakování

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2017

Predikát `fail`, negace

`fail`

- cíl, který vždy selže
- spolu s řezem: selhání při splnění podmínky
- lze jím implementovat negaci

```
1 notnum(X) :- number(X), !, fail.  
2 notnum(_).
```

Predikát `fail`, negace

`fail`

- cíl, který vždy selže
- spolu s řezem: selhání při splnění podmínky
- lze jím implementovat negaci

```
1 notnum(X) :- number(X), !, fail.  
2 notnum(_).
```

Negace (`\+`)

- negace jako neúspěch
- jen zkratka pro předchozí konstrukci
- doporučuje se používat jen na cíle s plně instanciovanými argumenty
- `\+` P vyžaduje konečné odvození P

```
1 notnum(X) :- \+ number(X).
```

Negace predikátu s proměnnými

```
1 primary(red).  
2 primary(green).  
3 primary(blue).  
4 composite(X) :-  
5     \+ primary(X).  
6 nice(yellow).  
7 nice(green).
```

Zpoza negace nelze vrátit substituci:

```
1 ?- composite(X).  
2 false.
```

Negace predikátu s proměnnými

```
1 primary(red).  
2 primary(green).  
3 primary(blue).  
4 composite(X) :-  
5     \+ primary(X).  
6 nice(yellow).  
7 nice(green).
```

```
?- composite(X).
```

Zpoza negace nelze vrátit substituci:

```
1 ?- composite(X).  
2 false.
```

Negace predikátu s proměnnými

```
1 primary(red).
2 primary(green).
3 primary(blue).
4 composite(X) :-
5     \+ primary(X).
6 nice(yellow).
7 nice(green).
```

```
?- composite(X).
    |
?- \+ primary(X).
```

Zpoza negace nelze vrátit substituci:

```
1 ?- composite(X).
2 false.
```

Negace predikátu s proměnnými

```
1 primary(red).
2 primary(green).
3 primary(blue).
4 composite(X) :-
5     \+ primary(X).
6 nice(yellow).
7 nice(green).
```

```
?- composite(X).
    |
?- \+ primary(X).
    |
?- primary(X), !, fail.
```

Zpoza negace nelze vrátit substituci:

```
1 ?- composite(X).
2 false.
```

Negace predikátu s proměnnými

```
1 primary(red).
2 primary(green).
3 primary(blue).
4 composite(X) :-
5     \+ primary(X).
6 nice(yellow).
7 nice(green).
```

```
?- composite(X).
    |
?- \+ primary(X).
    |
?- primary(X), !, fail.
    | X = red
    ?- !, fail.
```

Zpoza negace nelze vrátit substituci:

```
1 ?- composite(X).
2 false.
```


Negace predikátu s proměnnými

```
1 primary(red).
2 primary(green).
3 primary(blue).
4 composite(X) :-
5     \+ primary(X).
6 nice(yellow).
7 nice(green).
```

Zpoza negace nelze vrátit substituci:

```
1 ?- composite(X).
2 false.
```

```
?- composite(X).
    |
?- \+ primary(X).
    |
?- primary(X), !, fail.
    | X = red
?- !, fail.
    |
?- fail.
```

Negace predikátu s proměnnými

```
1 primary(red).
2 primary(green).
3 primary(blue).
4 composite(X) :-
5     \+ primary(X).
6 nice(yellow).
7 nice(green).
```

Zpoza negace nelze vrátit substituci:

```
1 ?- composite(X).
2 false.
```

```
?- composite(X).
    |
?- \+ primary(X).
    |
?- primary(X), !, fail.
    | X = red
?- !, fail.
    |
?- fail.
    |
fail
```

Negace predikátu s proměnnými

```
1 primary(red).  
2 primary(green).  
3 primary(blue).  
4 composite(X) :- \+ primary(X).  
5 nice(yellow).  
6 nice(green).
```

Kombinace negace a proměnných:

```
1 ?- nice(X), composite(X).  
2 X = yellow.  
  
1 ?- composite(X), nice(X).  
2 false.
```

Příklad 11.2.2: Pomocí negace a predikátu `member/2` naprogramujte predikáty `inNeither/3` a `inExactlyOne/3` takové, že

- `inNeither(X, XS, YS)` uspěje právě tehdy, když prvek `X` není v seznamu `XS` ani v seznamu `YS`;
- `inExactlyOne(X, XS, YS)` uspěje právě tehdy, když prvek `X` je v právě jednom ze seznamů `XS` a `YS`. Predikát `inExactlyOne` zkuste naprogramovat tak, aby uměl do volné proměnné `X` postupně unifikovat všechny takové prvky.

Příklad 12.3.1: V jazyce Haskell naprogramujte rekurzivní funkci `app`, která se chová jako knihovnní funkce `++`. Dále v jazyce Prolog naprogramujte odpovídající rekurzivní predikát `app/3`, který se chová jako knihovnní predikát `append/3`.

V jazyce Haskell naprogramujte funkci `concat`, která se chová jako knihovnní funkce `concat`. Ekvivalentní predikát `concat/2` napište i v jazyce Prolog. *Pro pokročilejší:* V obou implementacích `concat` zkuste použít tail-rekurzi.

Příklad 12.3.2: V jazyce Haskell naprogramujte funkci `listAvg`, která pro neprázdný seznam čísel vrátí jeho aritmetický průměr. V jazyce Prolog naprogramujte odpovídající predikát `listAvg/2`.

Pro pokročilejší: Zkuste v obou jazycích `listAvg` implementovat tak, aby došlo jen k jednomu průchodu seznamu. V jazyce Haskell můžete zkusit použít vhodný `fold`; v jazyce Prolog akumulátory.

Příklad 12.3.3: V jazyce Haskell naprogramujte funkci `removeDups`, která všechny opakované výskyty libovolného prvku bezprostředně po sobě nahradí pouze jedním výskytem. Například

```
1 removeDups [1,1,1,2,2,3,1,1,4,4] = [1,2,3,1,4]
```

V jazyce Prolog naprogramujte odpovídající predikát `removeDups/2`.

Příklad 12.3.4: V jazyce Haskell naprogramujte funkci `mergeSort`, která seřadí zadaný seznam pomocí algoritmu *Merge sort*. Může se vám hodit nejprve implementovat pomocné funkce `split` a `merge`.

V jazyce Prolog naprogramujte odpovídající predikát `mergeSort/2`. Zkuste vhodným způsobem použít řezu.

Příklad 12.3.5: V jazyce Haskell naprogramujte funkci `primeFactors`, která pro zadané číslo vrátí seznam čísel, která odpovídají jeho prvočíselnému rozkladu.

Například

```
1 primeFactors 2 = [2]
2 primeFactors 4 = [2,2]
3 primeFactors 6 = [2,3]
4 primeFactors 30 = [2,3,5]
5 primeFactors 40 = [2,2,2,5]
```

V jazyce Prolog naprogramujte odpovídající predikát `primeFactors/2`.

Příklad 12.3.6: V jazyce Haskell naprogramujte funkci `quickSort`, která seřadí zadaný seznam pomocí algoritmu *Quick sort*.

V jazyce Prolog naprogramujte odpovídající predikát `quickSort/2`. Zkuste vhodným způsobem použít řezu.